BT Rules: a radically inclusive open source community

Cefn Hoile, Matt Jubb, David Gowans, Jia-Yan Gu

***Abstract*—Consumers rely on a proliferation of personal electronic devices whose native capabilities are augmented by software vendors and complemented by online service providers. It is feasible to compose tailored behaviours through bespoke software, but non-specialists face prohibitive technical and conceptual obstacles when attempting to program. The BT Rules project aims to put the authoring and sharing of new digital behaviours within the grasp of all our customers. This paper tries to articulate the key obstacles, and outlines the cognitive, technical, and social strategies we are employing to overcome them.**

***Index Terms*—visual programming, end user programming, programming languages, context free grammars**

I. Introduction

TODAY'S consumers own and interactively control many digital systems. These include desktop, mobile phone, and PDA operating systems, the software which runs on these devices, such as iTunes or Firefox, and hosted services which customers access through their devices, such as Facebook, IMAP email, or RSS. These digital systems are all fundamentally programmable, but are rarely programmed by the consumers themselves.

The BT Rules project aims to make it possible for end-users without any previous experience of programming to author new behaviours for this broad range of systems. Our team has combined emerging techniques with established infrastructural components and software standards to provide a seamless authoring and deployment experience.

The result; a unified paradigm through which consumers can control and automate the behaviour of diverse digital resources, supported by a web-based visual editor and centrally hosted script execution environment.

In this paper, we outline our approach by contrast with existing software management and mashup approaches. We catalogue a number of obstacles which prevent customers from *control*, *comprehension* and *creativity* in the digital domain, and explain how these issues are addressed.

We report on user interviews which were undertaken at the outset of the project to establish users' level of comprehension of key concepts, willingness to define system behaviours and capacity to manipulate a whiteboard-based physical analogue of the digital system.

Lastly, we make a case for hosting a community of consumer software authors on our central platform - an open source community, outlining how specific interactions hosted by this central facility can help both novice and established authors to corral software complexity, and eliminate potential bugs.

II. Existing Work

Modern systems are frequently furnished with application programming interfaces (APIs), or provide a standards-compliant exposure of their functionality. For example, Facebook provides well documented methods to access profile information, and to trigger events on the Facebook platform, whilst web servers frequently expose their mail facilities via the IMAP4S protocol, or their filesystem storage via SFTP.

Despite this openness, the users rarely construct their own automated behaviour to manipulate their Facebook profile or manipulate their email. Typically, users rely on qualified programmers to define integrated behaviours in advance, since each system requires a distinct set of technical skills and experience to automate it effectively. Each can require a different programming language or development environment, as well as a different mechanism to deploy the authored behaviour to the target execution context. The comprehension of language syntax and the design of logical operations using this syntax can be complex and error prone.

Many attempts have been made to create visual programming environments which try to hide the complexity of defining computer code for specific domains[1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. However, few have attempted to resolve the problem of authoring source code for multiple languages and target environments with a single visual metaphor.

Our project draws heavily on existing work in visual programming metaphors undertaken by MIT's Lifelong Kindergarten Group over a number of years[11], and embodied in the 'Scratch' programming environment[12], [13] . Scratch enables ordinary users to construct simple event-driven programs by constructing sentences from natural language tiles whose label, color and shape provide important cognitive clues as to their syntax and semantics.

Although we share Scratch's visual programming metaphor, the BT Rules project is entirely different in its implementation and target domain. Scratch is intended as a pedagogical tool rather than an orchestration tool in its own right. It has been written as a desktop application which combines both an authoring environment and an execution context for authored scripts. It is oriented towards the domain of multimedia-driven applications.

In contrast, BT Rules incorporates a clean-room HTML+JavaScript implementation of a drag-and-drop tile-based authoring environment for arbitrary context-free grammars loaded at runtime. A compiler has been constructed which allows target code to be compiled from the graphical tree manipulated by the user into object code (such as JavaScript or Python) suitable for remote deployment to

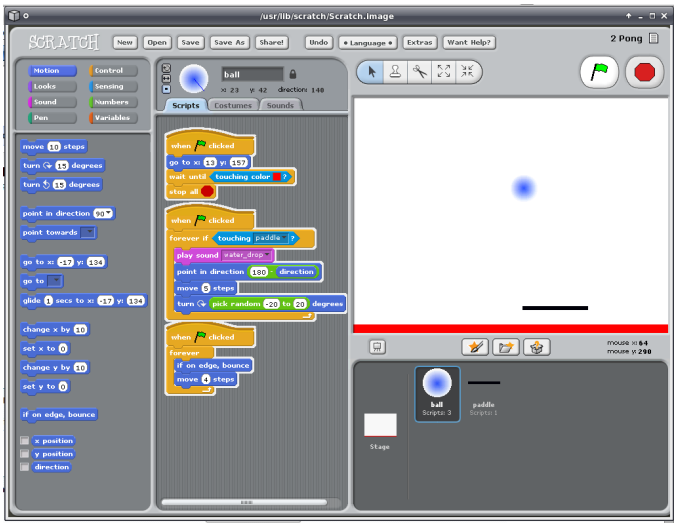


Figure 1. A Scratch script defining the ball behaviour for a game of Pong

various scripting environments. Multiple different domain-specific grammars can be transparently loaded during a typical authoring session. The graphical front end is backed by a distributed infrastructure comprising both server-side and client-side script hosting and coordinated by a wide-area type-safe messaging platform.

Our adoption of Scratch's tile-based programming metaphor enables us to avoid the shortcomings of traditional programming languages whilst unlocking their power. Today, users who are able to articulate the shortcomings of their devices' behaviour and even outline improved behaviours in natural language, find it impossible to author instructions which a computer can execute. Even professional programmers are typically able to code behaviours only for a limited subset of deployment environments in which they have expertise. They too face a steep learning curve to get to grips with a new set of tools and capabilities.

Most pre-existing approaches make sacrifices as they seek to *shield* the user from programming. Instead, we try to unlock the compositional richness which programming affords, but shield the user from unnecessary cognitive hurdles associated with two key phases*; constructing* valid code to represent a behaviour, and *deploying* code which implements that behaviour to the target device.

III. Consumer Control Paradigms

Non-programmers seeking a specific digital behaviour have few choices today.

They can explore the user manual, interactive menus and configuration options provided as part of a product or service, and hope that it is possible to achieve their chosen behaviour in this way. Alternatively they can explore the market for third party software, which may extend a system's capabilities to accommodate the chosen behaviour. Lastly they can attempt to author their own behaviour through a 'mashup'.

Our project falls into the latter category, but we believe there are important contrasts to be made with pre-existing work in all of these categories.

A. Manuals and Menus

In an ideal world, the creators of a device, application or service would have anticipated all the relevant behaviours and integrations with other devices which could possibly be desired by their future users.

Of course in practice, there are limitations on the amount of effort which can be invested in programming, design and testing. However, even if we discount this limitation there are additional, and probably more significant, limitations imposed by the presentation layer of a manual or menu; that technical writers and user interface designers have to employ labelling and organisational principles which are simple and approachable to the lay user. The more features which are crammed into the documented capabilities of a device and its configuration options, the less likely these features are to ever be encountered or understood. We expand on this issue in Section IV-A when we explore the difference in the expressive power of a combinatorial language by comparison with the 'parameter space' exposed to the user using traditional menu techniques.

The expressive richness of BT Rules scripts makes the case that system designers need not limit their systems' capabilities merely in order to make them comprehensible to the lowest common denominator user, assuming the choice of interface is correct.

B. Third Party Software Development

Although it is possible to commission a professional programmer to construct and maintain bespoke software tailored to a single user, costs are prohibitive. Consumers seeking an automated behaviour which is not available out-of-the-box typically rely on an industrial software production pipeline fed by marketing intelligence about bulk user requirements at one end, and supported by domain specialist programmers at the other.

If they are lucky, the pipeline will serve them well. Their *need will be identified*, it will be judged to be *cost-effective or desirable to supply* that market, a piece of *software will be written* which is fit for purpose and they will *discover the software*.

However, this strict set of criteria severely limits the likelihood of a customer encountering and acquiring digital behaviour which suits their requirements. The resource overhead of this pipeline is very high, incorporating marketing and usability professionals, accounting staff and programmers; and its capacity is consequently very limited. The more specialist their requirement, the smaller the chance of scarce resources being made available to satisfy it.

BT Rules overcomes this resource limitation by enabling the user themselves to resource the requirements capture, behaviour authoring and testing. The expensive pipeline collapses into a process which can be managed by single person, who also happens to be the end-user.

C. Mash-up and Script-hosting providers

A mash-up is an orchestration hosted by a provider who exposes various systems to a consistent middleware and

automation layer, using simple metaphors targeted to non-programmers.

A very large number of 'mash-up' systems have been launched by both start ups and major players in the IT industry in recent years. These include Yahoo Pipes[14], Microsoft Popfly[7], Tarpipe, IBM Mashup Centre[15]. These initiatives aim to provide authoring environments enabling their users to combine system functions from multiple different providers to create personalised applications which are 'greater than the sum of their parts'.

In common with the BT Rules project, some provide graphical tools with simple metaphors to construct orchestrations of those systems and orchestrations are hosted on the company's infrastructure. Also in common with our project, many are predicated on the exposure of system functions through public APIs and other standards, although it is worth noting that BT Rules is not *restricted* merely to control domains exposed through web-based eventing, since our imperative programming tools also allow the behaviour of a user's devices and software to be controlled.

Simpler propositions are available from Google App Engine[16], Heroku[17] and AppJet[18], who offer to host scripts for networked interactions, but without providing a graphical authoring environment, falling back on conventional programming approaches.

At first glance, it seems simple to define combinations of services using these tools. Many of these services offer a wiring metaphor which allows you to connect the output of one module to the input of another module. From the point of view of a user-programmer this dataflow is easy to construct and easy to comprehend. However, in practice, it is the logic hidden inside the individual blocks which provides the core functionality. It is a testament to the unwieldiness [2] of the data-flow programming approach that even in systems where the user is permitted to author the blocks themselves, they are invited to use a full-fledged programming language to do so.

A mashup system has a richer palette than the individual systems themselves provide. Data-flow relationships can combine behavioural blocks through instantaneous event passing rather than persistent parameterisation (see Section IV-A for more on this). However, in the final analysis, these mash-up providers face the same limitations as regular integrators - that skilled programmers must anticipate and implement the imperatively defined event-driven behaviours made available to users.

BT Rules aims to eliminate the dependence on traditional programming languages to define imperative behaviours, making it possible to extend true personalisation to non-programmers. This personalisation extends beyond web-based services to include direct control of the user's devices.

IV. Advantages of the Approach

The BT Rules infrastructure incorporates a wide variety of innovative technologies for managing user data, managing and hosting scripts at scale, but the primary focus of the project is the authoring facility.

Inspired by the Scratch programming language, our team has developed a novel form of programming infrastructure which allows an ordinary user to describe their preferred digital behaviour in plain English through a conventional web browser, then immediately deploy this 'program' directly to their chosen device, software or service within seconds.

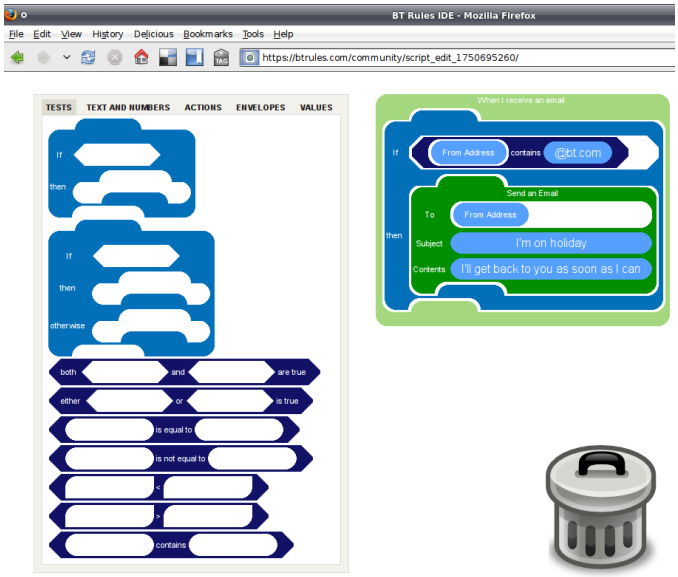


Figure 2. BT Rules user authoring an email handling program, showing script composed of tiles by drag and drop from a tabbed palette

In common with Scratch, the editing facility has been conceived to provide cues for valid program construction which are cognitively matched to the capabilities of everyday users.

The elements which authors use to create programs are graphical tiles annotated with English phrase fragments, whose shape corresponds with the shape of the slots where they may be used. Once all slots have been filled, the program is complete and valid and the phrases on each tile can be read from top left to bottom right to form an English sentence. The completed sentence describes the behaviour in human comprehensible terms.

The authoring environment is backed by a script hosting infrastructure. A version of the script is compiled from the sentence of graphical tiles into a suitable object language, such as JavaScript or Python. This compiled script is delivered to the target execution environment using the XMPP protocol.

BT Rules scripts deploy to a variety of targets. In some cases, the environment itself has the ability to host a scripting environment, such as the Microsoft Windows desktop, Firefox web browser or Symbian 60 Mobile phones. In other cases such as RSS, IMAP email, Google Calendar[19] and Google Translate[20], hosting resources for a script interpreter have to be centrally provisioned for that user, and the authored behaviour orchestrates a networked protocol. In a third case, a script interpreter installed on the user's own device acts as the orchestration engine, manipulating a locally installed device which cannot itself host code. This approach has been used to command a PC to issue control messages over a locally installed gateway to passive devices supporting the X10 home automation protocol [21] and is suitable for manipulating con-

sumer electronics by directly scripting an infrared transceiver, but indirectly scripting televisions, video and audio players.

We have proven the approach to define behaviours in real time for the Windows Desktop, Symbian Mobile Phone, Facebook, IMAP email, X10 home automation, Bluetooth devices, the Firefox web browser, RSS feeds, a variety of Google services and many other environments. In addition it has been proven for arbitrary combinations of these devices working in concert, offering the potential for consumers to use BT Rules to unify the components of their digital livestyle in a single coordination infrastructure. A subset of prototyped deployment 'contexts' were hardened and made available for a live trial launched in January 2009.

The next sections describe how the BT Rules unified control and automation strategy is superior to conventional approaches with respect to three key areas for non-technical users; *control* - their ability to command their digital world to do exactly what they wish, *comprehension* - their ability to understand the consequences of a given behaviour, and *creativity* - the payoff of a rich and flexible environment - that users can integrate different parts of their digital lifestyle inventively and, as we will see later in Section X, share these inventive steps with each other.

A. Control

Modern personal devices, and the servers which host consumer services are highly programmable by their very nature. However, the access provided for ordinary users to modify their behaviour is typically very narrow.

Instead of exposing the full capability of the system to the user, a tiny subset of the system's capability is selected as being the most relevant and meaningful to the average user, and menu and form controls are provided to allow the user to trigger hard-coded behaviour conceived in advance by technicians.

Occasionally these behaviours are controllable by granular settings, and behind the scenes, a partially specified program is 'completed' by the user's choice of parameters.

For example, mobile phone menus are frequently provided with 'profiles' allowing a small set of user-configured values to determine the phone's behaviour. Through profiles, the user is permitted to assign individual sound files to phone events. The user is not able to determine exactly what the phone will do in response to an inbound call from the full range of capabilities which the phone has (e.g. turn itself off, load a webpage, take a photograph, send an SMS message). Instead, users are restricted by the user interface, and the choice of 'partial program' they are allowed to populate through this interface, to simply selecting alert noises.

Equally, whilst the computer processor driving a banking server is quite capable of carrying out an aperiodic schedule of payments based on a dynamic calculation such as the number of working days remaining in the month, internet banking users may be provided instead with just a few parameters to control automated payments, such as: a fixed day of the month, a fixed amount, a fixed payee or the date of the last payment.

Following MIT's example of the Scratch programming language, the BT Rules project overcomes these limitations by exposing a comprehensive set of operational atoms from the underlying system as 'jigsaw' tiles bearing fragments of English sentences.

Strictly speaking, BT Rules is not a language, but a family of languages, each defined by its own grammar, The grammars define the operational atoms relevant to that domain, (such as the 'turn off', 'play sound' or 'take photograph' tiles which are available on a cellphone). A grammar also defines the relations these atoms must satisfy in order to create a valid program for the specific system.

Programmers will recognise BT Rules languages as type-safe, strictly constraining the values which can be passed into specific operations, but the users themselves are not expected to have experience of typed-languages. Instead, they can judge which combinations of tiles are valid through graphical cues such as shape-fitting and color-matching combined with their grammatical comprehension of the partial or completed sentences as they proceed.

Code deployed to control a system must of course satisfy the system's validity constraints. To satisfy this requirement, most integrators merely allow users to parameterise valid programs through a menu interface. Instead, we have designed the system so that the cognitive facilities required to author valid programs are found in the majority of the English-speaking population. Whenever a set of tiles fits together with no empty slots, the program constructed is guaranteed to compile to valid object code which can execute that behaviour on the target device.

The combined tiles can be read as a complete sentence which describes the observed behaviour which the digital system will display when it is deployed—critical for the user's *comprehension* of scripts they have authored as well as those they may acquire as defaults or through community sharing.

B. Comprehension

Even in the case where an individual behaviour has been anticipated by the vendor's integration team, it can be hard for the user to understand what the behaviour of the device, software or service will actually be. It is typically impossible to inspect the closed-source 'partial program' which has been constructed by the vendor and which is parameterised by the user-facing controls. Users must resort to reading the product manual, hoping that a technical writer was able to comprehensively describe the behaviour, or alternatively navigating the menu structure of the device in the hope that they will arrive at a self-explanatory control.

For example, in the case of a mobile phone profile, it may be that the 'Bugle' profile has been selected, a profile which triggers bugle sounds in response to declared alert conditions. However, at some point in the past, the user activated 'Silent Mode' in a meeting. This overrides the user's selection of all sounds for all alert types preventing any noises being generated. Since the 'Silent mode' option exists in a different part of the menu heirarchy; calls, messages or alarms can be missed despite the user's best efforts to comprehend and influence the behaviour of the device by changing profiles.

If a BT Rules approach was adopted to define the logic of the phone's behaviour, it would be possible for the user

to directly inspect and control the way a phone handles an incoming call event. If the vendor's default implementation visibly checks for 'silent mode' and suppresses the default sound, the user could read this as plain English. The user may then choose to deactivate silent mode.

It is worth observing at this point that the simplest form of BT Rules program is a script which immediately executes a single action. This means that the user may choose to bypass the vendor-supplied menu and interface altogether, and can adopt a BT Rules tile browsing approach to control every part of their device's configuration interactively. In this case, for example, they may create and execute a script which contains a single tile which reads <deactivate silent mode>.

More experienced users, who combine tiles together into more complex scripts may have avoided the problem altogether. Instead of permanently activating silent mode for a meeting, they could trigger a Rules-authored script which deactivates sounds, then waits for an hour before reactivating sounds again. One-off scripts of this kind can be authored and deployed immediately, and can incorporate the user's hard-won knowledge—that they frequently forget to reactivate the sounds.

Returning to the banking industry, the implementation of fraud rules in financial transaction systems are typically hidden. This can create unexpected behaviours and customer service issues which are costly to resolve. For example, the default behaviour of a credit card with a UK address may be to reject transactions originating in Taiwan. Since UK credit cards are rarely used in Taiwan, the cost of rejecting these transactions is considered to be low, compared to the instances of international credit card fraud which can be prevented. However, a customer travelling to Taiwan on a business trip may remain oblivious to this rule, only discovering when they arrive at the airport that they are unable to withdraw local currency.

Using a BT Rules approach, the customer could inspect and understand the details of the rules which the bank uses to keep their credit card transactions safe. If the bank chooses, their customers may be permitted to disable or modify the rules through an online interface, or alternatively they could contact a call centre to ask for them to be temporarily suspended to take account of their lifestyle. Additionally, they may choose to add their own additional fraud rules, which constrain valid transactions to a lifestyle envelope which *only they* can accurately define. For example; they may know for sure they never: buy anything online, or in the month of June, or spend more than $200. Enabling fraud systems to incorporate this knowledge from customers may assist fraud prevention by providing an early warning of invalid transactions—a canary in the mine.

This strategy—exposing the provider's own implementation of default behaviours to their customers and more radically, letting them disable or modify them—shares a great deal with the rationale for open source development. Both the user and programmer community can draw confidence from the fact that the system's behaviour can be inspected, and innovation from the community can be corralled for the benefit of the provider.

C. Creativity

The inventive capacities of everyday users, when they are able to freely express their own personalised behaviour, was qualified by evidence from user interviews, as reported in Section VI, and provides the engine for community sharing in the later Section X concerning community sharing. We have concentrated on the imperative control of the consumer digital lifestyle since it is likely to reach the widest possible audience of non-programmers, and hence unlock the greatest value.

The dominant paradigms outlined in Section III should be a concern for vendors who want their customers to upgrade to more and more capable devices, since it imposes an unnecessary limitation to the value users can gain from each new function. Users are effectively limited to the value which the integration team could imagine up front, have time to write partial programs for, and build user-friendly interfaces to control. The functionality unlocked by BT Rules frequently includes established system behaviours, as shown in the 'Out of office auto-reply' example in Figure 2—already a built-in capability in the Microsoft Exchange/Outlook mail platform[22]. However our system permits the users to invent their own unanticipated system behaviours, leapfrogging the sluggish development practices of a multi-year software release cycle. Additionally, if your installed mail platform happens not to provide a specific capability provided by a competitor, the user is in a position to address the shortcoming themselves in a matter of minutes.

Some of the commercial consequences of this shift in emphasis could be significant. Under the BT Rules model, vendors can concentrate on bundling core capabilities into competitively-priced or distinctively branded hardware, and leaving their customers to personalise their use of those capabilities. A platform vendor introducing an innovative feature into their core supported behaviours can expect those behaviours to be cloned on their competitor platforms by the end of the day. The implications of this model on the defensibility of software patents could also be important. Patent holders may find themselves in the unenviable position of seeking damages directly from consumers allegedly infringing their intellectual property rights for specific software behaviours.

Although this seems like a radical departure, it could be seen as a way of industrialising the process of user-driven innovations which has been recognised in many different industries.

V. Implementation

A. Language Definition

BT Rules programs are canonically represented as XML documents—trees of named 'elements' as the term is defined in the XML standard.

A single root element contains child elements, which may in turn contain further child elements. Leaf elements may be empty or may contain literal values. XML attributes are not currently used in our canonical script representation.

Each named XML element correlates with items in three different domains;

```
<?xml version="1.0" encoding="UTF-8"?>
<?oxygen RNGSchema="imap/grammars/receive.rng" type="xml"?>
<onemailreceive>
  <block>
    <ifonly>
      <eq>
        <emailfrom/>
        <strvalue>bttjpaper@cefn.com</strvalue>
      </eq>
      <block>
        <sendemail>
          <emailfrom/>
          <concat>
            <strvalue>I got your mail about </strvalue>
            <emailsubject/>
          </concat>
          <strvalue>Sorry, I'm on holiday. Back on the 19th</strvalue>
        </sendemail>
      </block>
    </ifonly>
  </block>
</onemailreceive>
```

Figure 3. XML canonical form for validation, storage (compare Figs 5,6)

- an explicit declaration of the element in an XML grammar (which defines all possible valid programs in their canonical XML form)
- named visual tiles or literal values in the DHTML graphical editor (permitting scripts to be authored by users in a browser)
- source code fragments in the final deployed script (permitting user authored scripts to be compiled)

We represent the context-free grammar which defines all possible valid XML 'programs' for a given digital system using a public-domain standard for this purpose known as RelaxNG. An example of the definition of the *if...else* construct is shown in in Figure 4. A canonical XML representation of a valid user-authored script validates against the grammar as specified in the RelaxNG standard. To bind together the three different domains, the RelaxNG grammars are specialised with BT-Rules-specific annotations. An example of the XML canonical form used by BT Rules is shown in Figure 3.

Graphical and textual annotations are used to represent the meaning of the events, values and actions available to the human being who is doing the authoring. These assets are used to translate a grammar into a drag-and-drop editor which can be used to author the XML canonical form of a program.

Code fragment annotations are read by a bespoke compiler to turn an XML program into interpretable code suitable to be executed in a given digital context. An example of the ECMAScript compiled form of a sample script is shown in Figure 6.

This resulting code typically takes the form of a script in a scripting language, but can in fact be any computer-readable form.

During important stages of the UI generation, editing and compilation process, the RelaxNG grammar structure is directly or indirectly queried for information related to the grammar structures, and to the annotations which embedded in that grammar structure.

Direct queries of the grammar and its annotations take place using Xpath in the context of an XSLT stylesheet or an XQuery program. The type ancestry of an individual tile is queried

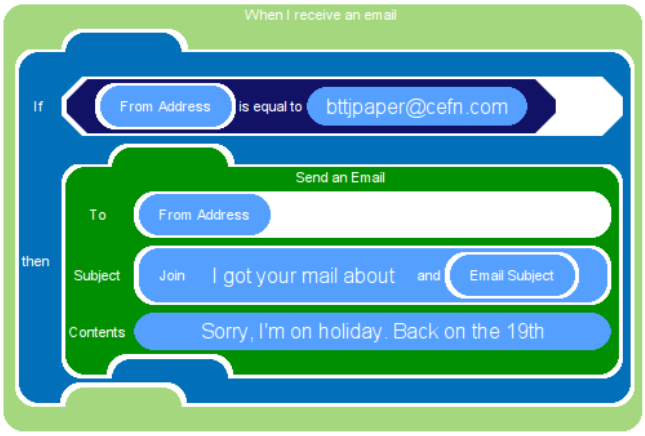


Figure 5. DHTML graphical form for authoring (compare Figs 3,6)

```
loadLibrary('core.js');
loadLibrary('imap.js');
addEventHandler('onEmailReceived',
      function(ev){
            ev.threadId = startScript();
            if((ev.from=='bttjpaper@cefn.com')){
                  email.send(
                        ev.from,
                        'I got your mail about + 'ev.subject,
                        'Sorry, I\'m on holiday. Back on the 19th'
                  );
            };
            finishScript(ev.threadId);
      }
);
```

Figure 6. ECMAScript compiled form for execution (compare Figs **??**,5)

whilst generating the editor to decide its color and shape. The human-readable text for that tile is retrieved from an annotation of the tile's definition in the grammar. The tile's definition is also annotated with computer-readable interpreter code which defines the actual behaviour 'implemented' by the tile. The DHTML graphical form for authoring can be seen in Figure 5.

Not all queries take place in an XPath-enabled environment. Therefore this information is frequently re-represented in a form suitable for a particular purpose, enabling an indirect query of the grammar. For example, a task-specific JSON (JavaScript Object Notation) representation of the grammar is generated. This makes it easy to query the grammar from JavaScript for real-time control in the HTML drag and drop editor. Suitably named SVG files for the complex graphical features (such as corners) are also generated from the drawing paths which annotate the grammar, allowing graphical packages to request visual assets without directly querying the grammar.

The ability to query the relationships between tiles in the grammar is important throughout the application. For example, where a specific tile does not have style annotations of its own, its type-membership is interrogated, and the type declaration is examined for style annotations. If that particular type is not annotated with style information, then the supertype

```
<define name="ifelse" b:fill="#0170B9">
  <element name="ifelse" b:desc="If the given bool is true, execute the first set of actions, otherwise execute the second." b:prejs="if(" b:postjs="}" b:prepy="if(" b:category='tests' >
    <ref name="boolexpr" b:pretext="If" b:postjs="){" b:desc="The condition you want to test when choosing which actions to run" b:postpy="):&#xA;"/>
    <ref name="block" b:pretext="then " b:postjs="}else{" b:desc="The actions to run if the conditions are true" b:prepy="" b:postpy="&#xA;else:&#xA;"/>
    <ref name="block" b:pretext="otherwise " b:desc="The actions to run if the conditions are false" b:prepy=""/>
  </element>
</define>
```

Figure 4. Example grammar declaration of the *if...else* construct, annotated with additional graphical and compiler metadata

declaration is examined, and so on.

Similarly, to support the editing interaction, the grammar can be queried using XPath to establish the candidate tiles for a specific slot. Where the slot defines a type criterion for its children, it is possible to craft an XPath expression programatically which will retrieve all possible candidate children, by querying the type hierarchy in the grammar file. The resulting candidates can be provided in a JSON form to the editing software. This pre-defined list accelerates the verification of valid completions as the user undertakes real-time modifications of the abstract syntax tree.

During compilation, after the XML document is validated against the grammar using the RelaxNG toolset, the XML document object model (equivalent to the abstract syntax tree) is traversed in lockstep with the grammar file, allowing the compiler to identify the proper production rules which satisfy each subtree as the traversal takes place. Once a matching production rule is found, the compiler can retrieve the source code annotations for that production rule, and hence compile each clause recursively into executable script.

B. Wide Area Messaging

Wide area messaging provides an infrastructure for individual digital systems to exchange messages with one another.

In our case, this permits the editing server, where you drag and drop blocks, to send newly edited behaviours to run on a user's device, and to manage those devices remotely using crafted messages. It also enables a user's devices to talk to each other, allowing the user to create behaviours which combine more than one device through argument passing and continuations, (although this interaction complexity is hidden in the graphical interface).

We employ a public standard called XMPP which defines XML message stanzas supporting an ordinary instant messaging payload, as well as a series of ratified community standards for more specialised messaging. However the simple instant messaging analogy is adequate to understand most of the requirements of our application. Each device authenticates itself and joins a chatroom which only hosts devices for that specific user. Devices can then securely communicate between each other and the central server using this messaging hub. The current message structures rely only on the generic instant messaging payload, repurposing these structures for a bespoke machine to machine communication protocol. This simplicity means that future authors of BT Rules interpreters on any platform should find it easy to accept new deployment requests, marshalled arguments and other management commands using mainstream XMPP libraries for their preferred language.

XMPP messages have a sender, optional receiver, subject, thread ID and text body. A small set of messages must be supported by any BT Rules interpreter to permit us to manipulate its behaviour remotely.

For example, a new code deployment message is indicated by a specific trusted *sender*, a specific *subject* string, a *thread ID* identifying the script and a *message body* containing the new code. A complementary undeployment message is indicated by a specific trusted *sender* and a specific *subject* string while the *thread ID* indicates the script to undeploy.

A remote function trigger is also recognised by the *subject* value, sends function name and arguments in the *message body* and passes a *thread ID* which should be referenced as the *thread ID* of any forwarding function call, and by any reply which continues the enclosing execution context. This trigger should cause the execution of a function with with specific arguments, and the reply contains the resultant 'stack' context, including any values changed by the script.

We use XMPP in a specialised way in which:

- Devices are assigned an identity allowing them to securely communicate with the server and with each other on behalf of a single user account with a single username and password
- User-authored scripts are compiled and dispatched from the editing server for a user's device to execute. We have demonstrated the use of both JavaScript and Python as target languages
- Structured data can be exchanged between devices using JSON, and consumed by user-authored scripts allowing users to pass data between their devices

The error-free marshalling and unmarshalling of structured data is handled through a form of dynamically authored grammar extension informally known as an 'envelope'.

Users can define a specific envelope by detailing the fields in the envelope—the names and types of values which it can carry. When the definition is finished, this causes the generation of a grammar extension. The new grammar defines type-safe actions to send the new envelope, and a new form of event which handles the receipt of such an envelope.

Subsequently, when editing scripts, a new action will be available inside all scripts—a send action for that type of envelope, with type-safe slots for the specific values. A corresponding event will be available for all devices, with typed symbols which access the named fields in the envelope.

Since all scripts are centrally hosted, compiled and deployed, in a future embodiment it will be possible for a send action to provide links to show the user where any corresponding envelope-receipt scripts exist in their device cloud. Equally, envelope-receipt scripts can provide links to corresponding send scripts.

Complex distributed errors, such as events which trigger themselves or each other in an endless loop, can be identified

using an XPath expression which inspects all the user's scripts and identifies the structural features of a loop. This pattern-matching approach does not aim to establish provable properties for the system, but it can alert the user to an error condition before deployment which may otherwise be difficult to identify.

VI. User Interviews

At the time of writing, the public-facing BT Rules trial environment has only been live for a few weeks and continues in a rapid iteration as we aim to open up the beta to a wider audience. It is therefore too early to report on live trial results. However, we are able to report on a series of structured interviews early on in the project in which subjects were invited to re-imagine their digital lifestyles, proposing novel functions combining existing digital capabilities—such as sensor events, audio playback, messaging and remote controls—into behaviours they considered useful.

Evidence was gathered from both direct questioning and self-reporting; and from observation of users attempting to design an example program with minimal supervision. It should be noted that the interviews were undertaken principally to inform early design decisions in the project. The sample size and its demographics do not provide statistically significant results suitable for user segmentation.

Initial interviews provided us with indications of the digital environment, lifestyle, motivations and personal goals of five different user sectors: children, students, young professionals, parent professionals and the retired. A minimal set of target domains were found to be relevant across the user sectors: Mobile, Desktop, Web Services, and Home Appliances. This information offered a basis for the construction of personas and detailed scenarios used as seed ideas where necessary in the final set of interviews.

We constructed a strict set of grammars representing the underlying capabilities of this set of devices, and fabricated a limited set of magnetic tiles corresponding to the language atoms of these grammars. The physical nature of the prototype allowed us to rapidly iterate design ideas for tiles, and even permit users to create their own tiles during the interview as they saw fit.

In the final set of interviews, we asked respondents to manipulate the magnetic tiles to construct novel digital behaviours which they considered useful from the available repertoire of tiles with as little supervision as possible. On the rare occasion that they were unable to conceive their own desired behaviour, they were prompted with seed ideas from the original persona-based analysis. We recorded the key stages of each composition made at each stage. We wished to establish in particular whether unsupervised users would be stimulated to:

- rethink their own devices as programmable
- conceive and articulate desired regimes
- construct grammatically correct programs

Information was also collated to find out whether:

- text labels were self-explanatory
- shape constraints were respected
- unexpected issues were consistently experienced

A pilot test was carried out to finalise the choice of supporting materials and verify that the test duration of one hour and data collection methods were adequate.

Each test subject was given five minutes at the start of a test to familiarise themselves with the tiles and the test environment. All tiles were laid out in groups reflecting the organising principles of the proposed IDE. Tiles were colour coded based on their tile type: Event, Boolean Expression, Condition, Control Statement, and Action Statement; and were grouped according to the target system controlled by the tile before the beginning of each test. At the start of the test, a minimal description of the prototype interface and its purpose as a means to configure an automated behaviour was given. Photographs were taken as the test subject progressed, recording key stages of script construction and other specific contributions from interviewees such as new tiles, tile relationships or organising strategies.

Iterative testing was undertaken using the magnetic tiles to verify and fine-tune graphical designs for the drag-and-drop IDE for the cognitive skills of non-programmers. To validate the tile designs, we observed users constructing digital behaviours from a pre-determined tileset. Capabilities that consumers wished to manipulate were catalogued to help us to design the preferred tilesets.

20 volunteer subjects were tested over the period of one week. Half of the volunteers were non-programmers with no programming experience and the other half were programmers. The ages of the test subjects ranged from 14 to 50.

It was preferred to allow users to construct their own personalised behaviours, combining their own chosen tiles from those provided; however, scenario examples of varying complexity were provided on request if respondents could not proceed without examples. A pen was given to the user to allow them to build more personalised behaviours by creating their own tiles, as well as allowing the user to communicate missing features that they considered important. Emphasis was given to conceiving and constructing a behaviour in a way the user considered intuitive. In the final IDE design, users are prevented from violating the normative grammar by the constraints of the interface. The use of a physical prototype allowed us to observe the contrast between the strict validation imposed by a domain specific grammar, and the user's natural tendencies.

Particular attention was given to the errors that users made, the questions that were asked by the test subject and the user's approach to building a program that could orchestrate a digital behaviour to span more than one target system. These observations led to two different outcomes in the IDE design. Firstly, new or improved interface cues were provided where everyday users encountered misleading ambiguities and created programs which could not execute. Secondly, new forms of distributed execution were implemented which better matched the users' intuitions.

A number of significant behaviours observed in the two different user groups are shown in the table above, and detailed below.

Table I
BEHAVIOURS OBSERVED WITH MAGNETIC TILES FROM A SMALL SAMPLE - 10 PROGRAMMERS AND 10 NON-PROGRAMMERS

User Behaviour	Non-Programmer	Programmer
Scattering Tiles	2	1
Grouped Tiles by Colour	2	1
Horizontal Construction	2	2
Event-Action Sequence (high level)	7 (4)	5 (1)
Event ▷ Condition	4	0
Multiple Events**	0	2
Control Statement: single	8	7
Control Statement: IFELSE	1	5
Repeated Sequence: IF	4	3
Repeated Sequence: IFELSE	0	10
Nested Control Statements	2	5
Event Inside an IF Statement	1	2
Simultaneous Processing	0	1
Story-Telling Sequence	5	0
High Level Sequences*	2	3
High Level Tile Input Fields	5	7
Shout-Hear: High Level	4	0
Shout-Hear: Message	5	6
Shout-Hear: Value(s)	1	3
End Tile: Middle of Sequence	0	3
End Tile: End of Sequence	2	1
New Tile: FOREACH	0	1
New Tile: Message Content	1	1
New Tile: New Variable	0	1

A. Tile Gathering

A number of users took advantage of the board space to reorganise the tiles to suit their tastes. They began by picking up and scattering tiles over the board as part of their exploratory process, with groups emerging as they formed hypotheses about tile similarity, relatedness and relevance to their own goals. Collection of tiles by colour was observed, indicating that these visual cues were being recognised and their relevance assumed, especially among non-programmers.

B. Syntactic Non-Conformance

Shapes are assigned to tiles to reflect the valid combinations they can form with others. For example, actions which can be chained together are provided with a leading groove and a trailing tongue to indicate they can be placed in series, and branching statements such as *while* and *if* have a space which is distinctively shaped to accept a boolean parameter representing the condition which is tested as part of the script's control flow. As explained earlier, the interactive drag and drop interface constrains the user to conform strictly to a grammar and these visual cues help them to comprehend the constructions which the system will accept, and which can be compiled to a syntactically valid program. In the physical prototype, however, users were at liberty to ignore the shape cues, which they did frequently.

1) Horizontal sentence construction:

> "Initially I didn't even look at the shapes, when you code, you don't see shapes"—Programmer
>
> "so to word it differently then"—Non-Programmer when prompted to follow the jigsaw metaphor

We observed the construction of many programs as sentences laid out horizontally, rather than chained together vertically, ignoring the connectivity cues. However, users who were being led by textual information initially were able to quickly learn how to build in a vertical direction to respond to shape. A number of users demanded examples to follow to help them understand the shapes before proceeding.

2) When vs If; Event vs Boolean; Imperative vs Declarative: Another frequent violation of the shape-based syntactic constraints illustrated a confusion between a declarative or imperative mode of sentence interpretation, further complicated by an inherent duplication in the example tileset.

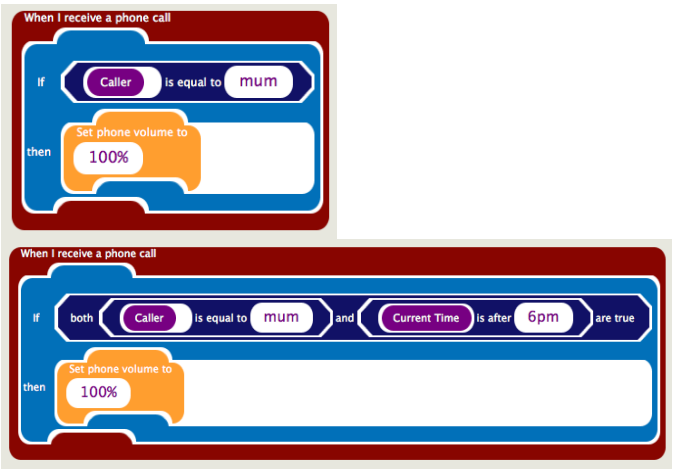


Figure 7. Imperative call handling scripts

Declarative and imperative languages correspond to different kinds of engine used to evaluate code in a computer. In one mode, statements are made about preconditions and postconditions, expressed as states of 'the world', and the computer detects when a given world state is satisfied (for example through propositional logic), and how to cause a given world state to be achieved (through planning).

The interpreters we have chosen to target with BT Rules are much simpler, and avoid the challenge (or impossibility) of creating a comprehensive and usable world model to cover the broad set of domains for which our users are authoring. So far, BT Rules target environments are merely imperative engines, in which an instantaneous event triggers a user-authored function, composed from a series of steps explicitly defined by the user. Imperative scripts in Python and JavaScript (the languages supported today by BT Rules) combine control flow statements and operations which can be trivially evaluated in sequence using the conventions of a stack pointer.

Nevertheless, non-programmers frequently tried to initiate scripts with propositions, co-opting the tiles intended for boolean logic in order to define a world state which they wanted to treat as an event. The contrast in sentence structure and semantics can be illustrated by the following fabricated examples. The first cannot be implemented by BT Rules imperative model, the second can.

- *'If my mum calls after 6pm make sure I hear it'* ; a propositional statement connecting world states together—suitable only if a comprehensive model and planner is maintained for the user's digital world, including whether they've heard the call!

- *'When I receive a call, if it's my mum and the time is after 6pm, set the ringtone volume to 100%'* ; a pre-defined atomic event triggering an action conditionally based on a compound boolean test—suitable for term-for-term translation and execution in an imperative script interpreter as seen in VI-B2

In a number of cases, users made the reverse mistake, employing a combination of *when* event tiles in an attempt to define a world-state, with one user combining an instant message receipt event with a time of day event, and a computer idleness event (see FigureVI-B2). When authoring declarative propositions about states it is possible to define compound states in this way. However it is not possible to author imperative event handlers combinatorially since each is instantaneous and connected to an independent trigger.

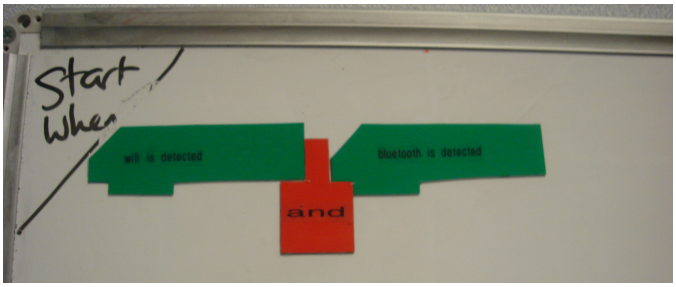


Figure 8. An attempt to define a compound event

Confusingly, in most cases, event and boolean tiles are *both* required for users to author rich scripts which take account of a given state: '*computer goes idle*' (event) indicates a transition between two states and can trigger the execution of a script, but scripts are also free to test the '*computer is idle*' (boolean) state during their execution to control conditional branching.

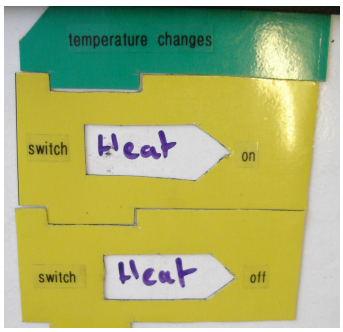


Figure 9. A story-telling script which describes an unintended behaviour

The discovery that half of the non-programmers adopted a 'story-telling' mode, conflating events and conditions ('*when*' versus '*if*') and drifting from imperative to declarative mode is a cause for concern, although this was not observed in the subjects who were programmers, suggesting an imperative mode may be learned by experience.

C. Semantic Non-Conformance

Users were sometimes observed to create valid programs quite explicitly described a behaviour which we believe they did not intend, as in the story-telling example shown in Figure 9. They may be surprised in the live system when the computer faithfully executes their instructions. In this case, whenever the detected temperature changed, the heating would be turned on before being immediately turned off within microseconds.

In other examples, automated, event-driven scripts were authored describing actions which the user may have better triggered manually either by interacting with a device and triggering its behaviour, or by using an '*When deployed*' event, which executes the instructions only once when the user chooses to deploy the script. We believe the example shown in Figure 10 was intended to send a thank-you email to Richard the next time the user was in contact with him. However, by this logic the thank-you email sent to Richard is sent whenever the user receives an email from anyone—probably many times a day—a behaviour likely to undermine Richard's appreciation. Note also the three redundant tiles, to

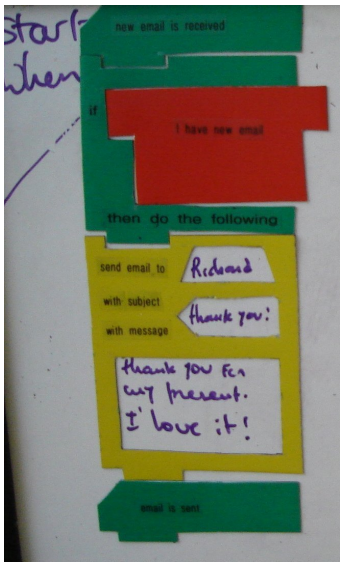


Figure 10. A script which expresses an action better triggered interactively

check if *I have new email* (this script is already triggered by a new email) and the use of the *email is sent* event tile to provide a redundant, action specific 'end tile', as outlined in the next section.

D. Non-Existent Tiles

Test subjects were able to create their own tiles at will, either by writing on the board surface, or by annotating a blank tile. A particularly interesting behaviour was observed independently a number of times. From the point of view of an imperative scripting language, the end of the script is the last statement in the script. Nevertheless, users frequently added an 'End' tile to satisfy themselves that the script had indeed ended (such as that shown in Figure 11.

A *for each* tile which allows a block of statements to be executed for each of a series of values in sequence was introduced by some users. Specific tiles were frequently introduced by programmers to create named variables, assign and retrieve values from them. Other ad-hoc tiles permitted value

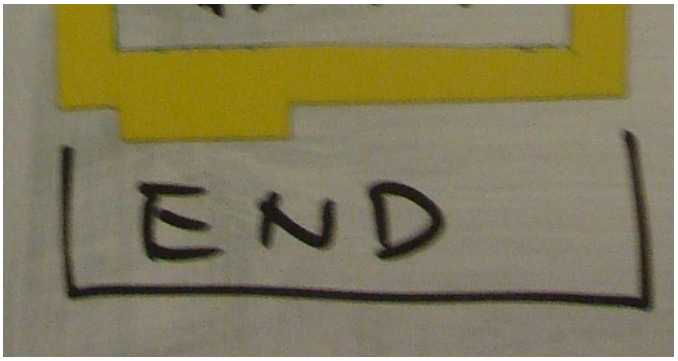


Figure 11. Spontaneous introduction of END tile by test subject

marshalling and unmarshalling from one script to another so that parameters could be passed between different devices.

Test subjects sometimes personalized their programmed behaviours by using high level inputs such as “me” in place of raw data such as the phone number that is associated with “me”. We intend to provide facilities for users to dynamically extend grammars with extra personalised value tiles which can be incorporated in scripts alongside the pre-defined tileset.

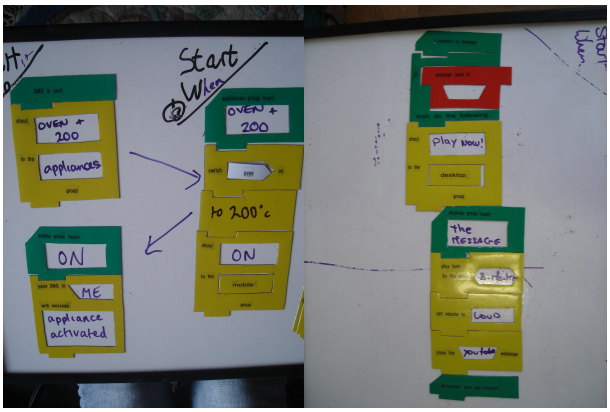
E. Additional Interface Features

We observed a set of behaviours suggesting cues which could be natural for everyday users in the final interface.

Users chose to lay out conceptually separate action sequences which are triggered by a single event in a horizontal fashion, with each sequence running in parallel, even though they may be technically required to run in sequence.

> “I want to have things laid out horizontally so that fragments happen from the same event”

Users tended to position the corresponding *shout* and *hear* tiles, (which allow devices to notify each other of events), in a way which was visually and vertically connected, suggesting sequenced execution, despite the different devices in which each script would execute (see Figure 12 . This suggests that the separation of code blocks according to the executing device may not be a preferred interface for these users.



(a) Connecting lines (b) Direct sequence

Figure 12. Shout and hear sequences visually connected

VII. Drag and Drop Interface

The core facilities of devices, software and services can be understood in terms of four fundamental building blocks.

Events take place in the context of a digital system, and can trigger behaviours which you define. They are the empty container which a user encounters when they begin to edit a behaviour. All behaviours are ‘contained’ within events, and are defined in terms of actions, values and logic.

The simplest event is *‘when deployed’* which simply executes the behaviour immediately on the target device. More complex events trigger a behaviour according to a change of state of the digital system, such as ‘*when I log in*’. Events may incorporate parameters, or be triggered by time and date conditions instead of system state, such as ‘*every 5 working days before the end of the month*’.

Actions are the blocks which from which users compose sequential behaviours. They can expose an operation from the underlying capabilities of the digital system, for example ‘*send an SMS*’, ‘*deactivate the screensaver*’ and can sometimes take parameters of their own, for example, ‘*set the volume to X*’ where X is a percentage. Atomic actions may also be embedded inside ‘flow control’ actions, for example [if [[today] is [friday]] [deactivate the screensaver]] where the ‘if’ action executes another embedded action conditionally, or [[play [beep.mp3]] [4] times] where an action repeats an embedded action in a looping construct.

Logic exposes the reasoning and value manipulation capability of the device. These are typically functions which transform one or more values into another value. This includes generic mathematical and boolean expressions such as ‘NOT(X)’, ‘OR(X,Y)’, and ‘AND(X,Y)’. Logic expressions may expose a transformational capability of the digital system (e.g. [lookup telephone number for [Chris Smith]] which takes in the name of a contact, and returns a telephone number, and exposes the capability of an address book or directory system.

Values are the data atoms which can be used as parameters to events, (e.g. every [friday]), to actions, (e.g. [play voicemails received on [friday]]), and to logic expressions (e.g. if [[day of the week] equals [friday]]).

A. Interactive UI Behaviour

Interactive UI Techniques are used to create an editing interface for users to construct valid programs. It provides real time feedback as they edit, helping them discover the constructions available and avoid programming errors.

In our current implementation all BT Rules programs are event listeners, and the unique items of information associated with an event are made available to the user as strictly-typed tiles which appear when editing their own handler for a chosen event.

In order to construct a program, users follow color, shape and text cues of tiles which have slots for other tiles to be inserted. These tiles are dragged and dropped inside each other (descendants) and beside each other (siblings) to form a hierarchy known as an Abstract Syntax Tree.

Tabbed, labelled palettes contain different groups of tiles. Tiles may be grouped according to conceptual similarity such

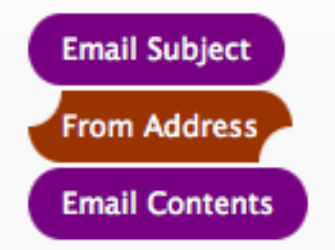


Figure 13. Event-specific typed values are made available to the user

as 'actions' and 'tests', or grammatical relationships such as strict typing, e.g. tiles which evaluate to a 'date' type.

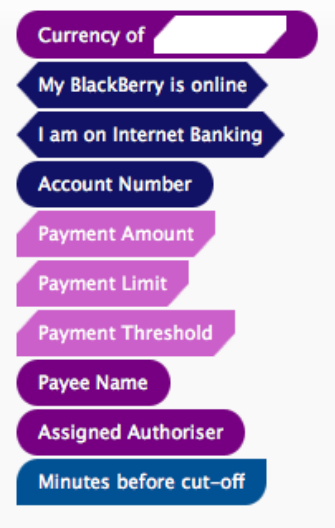


Figure 14. An example palette for a banking application

Individual tiles carry phrase fragments in a natural language. A completed program is typically human readable as a valid English sentence by concatenating these phrase fragments. 'Literal' values which are not calculated using other tiles, can be entered in tile 'slots' by users using the keyboard.

Each interactive UI technique relies on reading information from the grammar directly or indirectly in real time to examine valid completions for a specific editing step. Although the set of possible programs is infinite, at any stage of the editing process and for each tile slot there are a very small set of valid modifications which can take place, and these can be drawn to the attention of the user.

A given slot in a tile requires a descendant which conforms to certain criteria. The interface can positively indicate to the user the tiles or character sequences which are valid for that particular slot both statically and dynamically. These techniques offer a graphical analogue to the techniques of compile-time type checking and autocompletion as found in interactive editors for type-safe languages such as Java.

Static cues for valid completions (i.e. type membership) are provided using color and shape. The UI is designed so that tiles of specific colors fit into slots of matching colors. Equally tiles of specific shapes fit into correspondingly-shaped slots. Graphical features such as RGB color recipes and SVG drawing paths for tile corners are embedded as annotations in the type declarations found in the grammar. This ensures consistency between the language validation and the graphical representation. Whenever the grammar is modified, or new tiles are introduced, the visual representation of that grammar is guaranteed to keep pace.

Dynamic cues can also be offered in response to editing actions initiated by the user, for example selecting a slot, selecting or dragging a tile, or typing a value in a slot. Potential tiles may be highlighted to the user when they select a slot. Equally, potential slots may be highlighted to the user when they select a tile.

In future embodiments, other knowledge may be used to prioritise and order the tiles alerted to the user, such as the popularity of the tile, or the expected tile which is most likely given the editing context. This extra information can be gathered through the analysis of code which is centrally hosted on users' behalf on our editing server.

When users enter character data from the keyboard, they can get instant feedback if the data entered does not conform to the proper specification. Behind the scenes, the editor is analysing the characters to establish what valid types may be satisfied by the user's input, and assigns the most narrow type available.

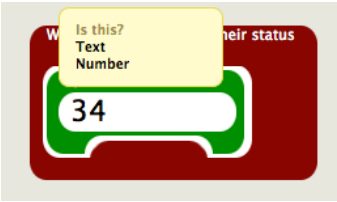


Figure 15. Interactive cues can ensure type-safe compilation

As a consequence of the strict typing of user input, when data is encountered in a script, or sent in a message, suitable metadata is associated. For example, the user input *http://cefn.com* can be dispatched in an envelope to another device with type metadata indicating it's a URL, rather than just a string of characters. This then permits the received value to be used with type-safe operations which themselves require a URL. The user never has to explicitly mark up their character input and implicit mechanisms are able to be used throughout.

VIII. Project Status

We have incorporated features arising from test users' feedback in response to the physical prototype; and in the graphical tool, interactive techniques are used to indicate correctness of tile position, providing users with direct feedback as outlined in VII.

Initial user testing with physical tiles was followed by a short trial of the prototype BT Rules graphical editor with a trial user group authoring Greasemonkey [23] user scripts for Firefox. The second live trial of BT Rules is now underway, providing a web-based front end to define behaviours which combine a variety of network-hosted services, including GMail, Twitter, Google Translate, and location tracking.

For the second trial, we have eliminated the *shout* and *hear* metaphor and instead permit users to employ tiles from

multiple devices into a single sequential script, implicitly defining messaging relationships between them.

Figure 16. Combining RSS and Twitter in a single seamless script

Type-safe variables can be created permitting devices to seamlessly share data within a script. Users' dissatisfaction with implicit *end* blocks has been met by graphically bracketing event handling scripts in a consistent way with other code blocks, equivalent to curly brackets in C style languages.

Experimental systems controlled by BT Rules in private testing in the labs include Retail banking systems, Nokia mobile phones running Symbian 60, X10 home automation tools[21] and Windows desktops. A *run immediately* event has been implemented for users who want to take advantage of the authoring framework to create one-shot operations. A prototype simulation layer exists which allows users to execute the scripts they author in an imperatively controlled artificial world and verify that the logic they have defined is indeed as they expect.

IX. Future Work

> I only understood what to do after I finished the first sequence

As explored in this paper, BT Rules offers an extremely user-friendly interface for both programmers and non-programmers to manage their digital devices, but users still desire guidance when creating new behaviours. A wide selection of potential devices to program or a blank IDE screen can be daunting for the users and they will look to the BT Rules infrastructure for inspiration and support.

We can respond to users' requests for language guidance in two ways, firstly by providing easily encountered illustrative examples, and secondly by hosting a community who can construct examples which are relevant to their own lifestyles and facilitate the sharing of examples between peers.

By providing example behaviours which are easy to find, understand and use, we eliminate one of the key hurdles to users wishing to use the system. Rather than having to build a script from scratch, they may choose a pre-existing one from a repository which they can edit to achieve the behaviour they desire. The pre-defined scripts bring the benefits of:

- illustrating the techniques and structures used in BT Rules
- demonstrating the capabilities of the language
- providing inspiration for the user's own scripts
- encouraging the uptake of new devices, software or services

Once a suitable repository of example scripts is in place, and users are familiar with adapting them for their own use, it is a logical progression to move to a community where users may share their scripts amongst each other. We intend to build a rich framework to allow this community to grow organically around the capabilities of the system. Users will be given a variety of tools to allow them not only to share scripts to the community and to adopt other scripts but to also share knowledge and best practice through code fragments, design patterns and annotations and comments on specific language tiles.

Through the use of a single hosting platform for the scripts, histories and ancestry trees may be maintained allowing users to examine the origins, and future reuse of their scripts, further enhancing their ability to contribute to the community, through inspiring others' creations and learning from reuse of their own. We expect ad-hoc communities may emerge around groups of scripts rather than formally defined projects, as is common in traditional Open Source communities.

An area which will be examined in depth is that of 'unusable tiles'. This can take two forms and both could drive interesting user behaviours. Firstly, the facilities will be provided to enable a user to build a script for a device which they do not yet own. A user may not have an iPhone, but may choose to develop a behaviour which includes the iPhone as one of the devices. This enables them to experiment with the possibilities of the device without the investment to buy one. We envisage this may drive higher uptake of some devices if users are able to appreciate their capabilities before purchase.

Conversely, it may be possible to give users the ability to suggest definitions of tiles themselves, where the behaviour does not yet exist. They may suggest, for example, capabilities or even full devices which they would like to be able to control, which could then be evaluated by the developer community as to whether they are feasible and desirable to implement. A key challenge with this approach will be to develop a suggestion system which does not become overrun by users requesting duplicate functionality, behaviour they could define themselves in a script, or small variations on existing functions.

We intend to investigate in depth the process a user goes through in creating a script, from inception through to the deployment and 'debugging' stages. Through the collaborative website hosting the users' scripts, we can examine and track a script's history, allowing in-depth analysis of the users' cognitive processes and design decisions. We intend to examine the choices of code reuse (from the community, from a user's own existing scripts of from pre-made examples); the style of use (many or few scripts, simple or complex behaviours) and the level of interaction between users in the community.

A key area we wish to address in the future is methods to convey to users the operation of their scripts and behaviours without them actually running. This becomes increasingly important as chargeable services, such as SMS messaging, are introduced, where a mistake by a user could result in them receiving a large and unexpected bill. We will investigate various approaches to simulate the user's 'world' and to illustrate the outcomes of their behaviours. There are significant challenges here, particularly in the area of 'out of band' effects where,

for example scripts are triggered indirectly by the effects of another (e.g. turning the house lights on automatically when it gets dark may inadvertently trigger behaviours intended for when the user returns home from work (such as turning on the oven), with potentially undesirable consequences!)

An interesting area we hope to investigate is that of business models for a community site of this form. The flexibility and power of the platform offers a variety of avenues for revenue generation including traditional channels such as single purchase or subscription payments. More interesting opportunities arise, though, from the possibility of building a marketplace for users, where scripts have a 'value' and can be traded and shared amongst users. Alternatively, models around the generation of revenue for device providers (such as introducing chargable services, such as SMS messaging), may be key revenue drivers through increased uptake of the service due to its programmability. It may be sufficient, though, that the community would drive increased usage of the services, increasing the goodwill of participating companies, without the need to bill users for the services.

By introducing a new programming paradigm and giving extensive control to users, BT Rules opens a wide range of possibilities, but in doing so introduces new issues and raises key questions which must be addressed. The team will focus on these as the system is developed, providing not only insight into how people use BT Rules, but also into the behaviour of 'non-programmers' when given the tools to create and share digital behaviours.

X. BT Rules and the Open Source Community

The current BT Rules implementation incorporates many open source technologies, including the following well-known open source libraries and utilities:

- Webserver technologies; Apache2 HTTP Daemon, Jetty, Apache Cocoon
- Database technologies; MySQL, Saxon XSLT transformer, eXist Native XML Database
- Messaging technologies; Openfire, Jabberpy

However, BT Rules users themselves represent a radically inclusive open source community in which the thresholds to contributing new code, and understanding existing code have been substantially lowered, enabling a much broader group of users to engage in programming and code-sharing. This contrasts with today's open source community in which new code and bugfixes can only be understood and contributed by domain specialists.

We plan to host a community of users engaging in this creative activity in a collaborative website. The sharing of best practice and new ideas within such a community will stimulate the adoption of the BT Rules authoring facility by specific demographic and industry segments, as well as the takeup of the products and services underlying popular community-authored behaviours.

During the trial, we hope to facilitate the creation of specialised peer groups through which individuals' script designs can be shared with other users with common concerns, for example users sharing the the same profession, with similar domestic devices or communications tastes.

XI. Conclusions

Writing and editing programs using traditional text editors is too difficult for many users, and even skilled programmers find that they make many mistakes of intent and comprehension during the process. So many different languages exist for different contexts, with their own syntax, conventions, build and deployment process that it is a challenge for an IT professionals to master more than a few of them.

The BT Rules system can be used to package almost any language, API or set of conventions in a simple graphical presentation which is suitable for non-programmers, growing the market for all forms of digital systems by removing key cognitive obstacles. Our approach eliminates the need for product manuals or computer language references to understand existing functionality. It also bypasses the need for specialist software delivery teams to deliver new functionality for a device, software or service.

Programs are written and understood as ordinary English sentences by dragging words and phrases together. The colour, shape, and graphical interactions between phrase fragments provide easily understood clues to authors as to the possible ways they can be combined.

At present we limit our scope to the imperative context-free languages that are expressable using a RelaxNG schema, but the same toolset can be employed to author domain specific declarative languages or broadened to richer grammar structures. It is feasible for the same graphical metaphor to incorporate and extend SQL and XPath. Language schemata with co-occurrence constraints, non-regular constraints, and inter-document constraints, such as those expressible in Schematron may also be accommodated in future versions.

Programmers rarely have the expertise to program multiple diverse systems. Even programmers with a very wide range of target systems find it hard to construct, deploy and test digital behaviours to all these systems rapidly, especially where an application combines more than one. For these reasons our toolset may also be of relevance to IT professionals.

One of the most encouraging comments on the BT Rules approach came from a subject during testing of the physical prototype.

> "for programming, I wouldn't use it, but for defining automatic behaviours in the home, it's a nice and simple way for fitting things together"

The irony is that programming is exactly what the subject was engaged in.

We believe that by making programming so seamless that consumers don't realise they are doing it, we can address a critical obstacle for consumers to unlock the value of their digital estate benefiting both vendors and customers and carving out a new niche for BT as a trusted partner in the management of our increasingly complex digital lives.

References

[1] G. C. Pierson, "Code maintenance and design for a visual programming language graphical user interface."
[2] O. Beletski, "End User Mashup Programming Environments,"
[3] S. Chang, *Principles of visual programming systems*. Prentice Hall Englewood Cliffs, NJ, 1990.

[4] S. Chang, *Visual Languages and Visual Programming*. Perseus Publishing, 1990.
[5] D. N. Chorafas, *Visual Programming Technology: Covers VRML, Java*. pub-MCGRAW-HILL:adr: McGraw-Hill, 1997.
[6] E. Glinert, *Visual Programming Environments: Applications and Issues*. IEEE Computer Society Press Los Alamitos, CA, USA, 1990.
[7] E. Griffin, "Foundations of Popfly: Rapid Mashup Development (Foundations),"
[8] M. Norrie, "PIM Meets Web 2.0,"
[9] N. Shu, *Visual programming*. Van Nostrand Reinhold Co. New York, NY, USA, 1988.
[10] J. Wong and I. Hong, "Marmite: Towards End-User Programming for the Web," in *Proc. of CHI*, vol. 7, 2007.
[11] M. Resnick, S. Ocko, and M. I. of Technology, *LEGO/Logo–learning Through and about Design*. Epistemology and Learning Group, MIT Media Laboratory, 1990.
[12] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick, "Scratch: A sneak preview," in *Second International Conference on Creating, Connecting, and Collaborating through Computing*, pp. 104–109, 2004.
[13] A. Monroy-Hernández and M. Resnick, "Empowering kids to create and share programmable media," 2008.
[14] M. Pruett, *Yahoo! pipes*. O'Reilly, 2007.
[15] I. Works, "Composite applications–Business Mashups," 2006.
[16] "Google app engine http://code.google.com/appengine/."
[17] "Heroku (ruby authoring and hosting infrastructure) http://heroku.com."
[18] "Appjet (authoring and hosting infrastructure) http://appjet.com/."
[19] "Google calendar http://www.google.com/calendar."
[20] "Google translate http://www.google.com/translate."
[21] K. Boone, "Using X10 for home automation."
[22] A. Gibbons and P. Singh, "How to Use Outlook Out of Office Assistant," 2002.
[23] M. Pilgrim, *Greasemonkey Hacks: Tips & Tools for Remixing the Web with Firefox (Hacks)*. O'Reilly Media, Inc., 2005.